Department of Electronic and Telecommunication Engineering

University of Moratuwa

EN3150 – Pattern Recognition

Assignment 02:

Learning from data and related challenges and classification

Anjula M.K

210368V

Date- 30/09/2024

Contents

1.Logistic regression

1.1 Generate Data

Generate Data using listing 1.

1.2 Purpose of Label Encoding 'species'

- The purpose of this line code is that it encodes the ‘species’ column which contains categorical data into numerical values using the LabelEncoder. LabelEncoder transforms these categorical values ('Adelie' and 'Chinstrap') into numerical labels as machine learning models like Logistic Regression can't directly work with categorical data.

- This encoding approach not only aids in model training but also enhances interpretability. For instance, models such as decision trees or logistic regression can highlight the features that have the greatest influence on predicting each species. Furthermore, encoding enables more effective data analysis and visualization, helping to cluster and investigate the relationships between species and their anatomical characteristics.

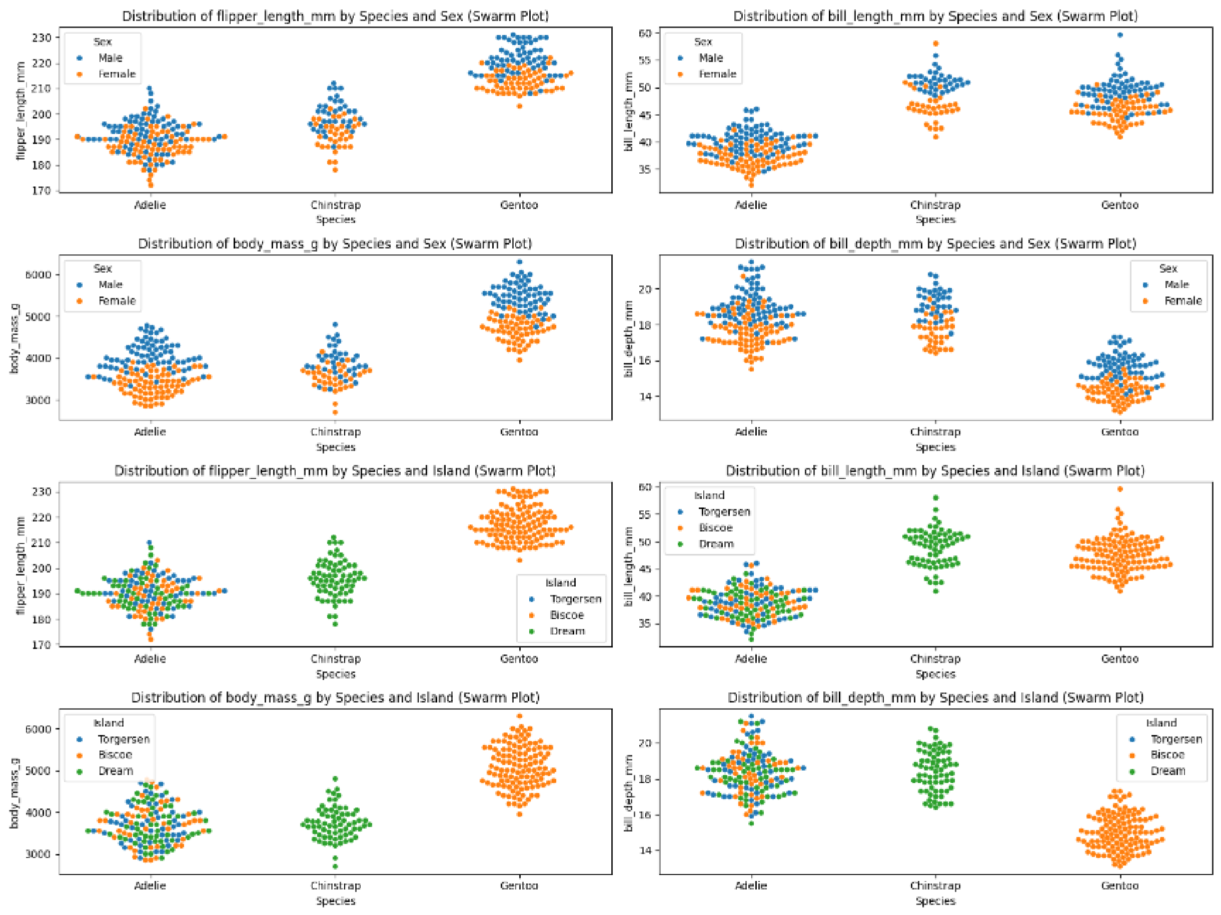
1.3 Purpose of Dropping 'species', 'island', and 'sex'

- This code line removes the columns species, island, and sex from the DataFrame to create the feature set (X). The species column is target variable. Therefore, it should be excluded from the feature set (X) used for training the model. Including it would result in data leakage, allowing the model to access the target during training and leading to falsely elevated performance metrics. The island and sex columns are dropped because these features may not be useful or suitable for the model in their current form.

- This step streamlines the dataset by keeping only the essential columns for modeling, such as numerical measurements, while excluding descriptive fields like 'island' and 'sex'.

1.4 Why Not Use 'island' and 'sex' Features?

- The features "island" and "sex" are likely not essential for differentiating between species in this dataset. Instead, focusing on physical traits such as bill length, bill depth, flipper length, and body mass tends to yield more relevant insights for classification. By visualizing the distribution of these features across species and categorizing them by "island" and "sex," it becomes apparent that these categorical variables do not provide meaningful distinctions between species. Therefore, "island" and "sex" could add unnecessary complexity to the analysis and could potentially be excluded from the model without compromising its predictive accuracy.



1.5 Training a Logistic Regression Model

Train a logistic regression model.

1.6 Usage of random state = 42

- The random_state=42 argument ensures that the splitting of the dataset into training and testing sets is reproducible. This allows for consistent results every time when run the code. By setting this seed, you make sure the train-test split will always be the same, which is helpful for debugging and comparison of results. By setting this value to a constant, it will make sure the train-test split will always be the same, which is helpful for debugging and comparison of results.

1.7 Reasons for Low Accuracy and Poor Saga Solver Performance

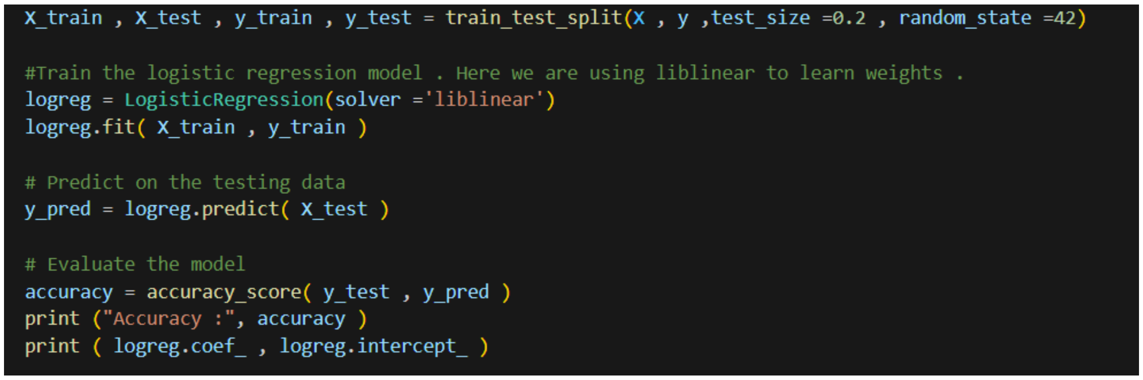
Low in Accuracy

- Accuracy will be low as not all features in the dataset may be useful for distinguishing between species. The features left in the dataset after dropping species, island, and sex might not be strong enough to differentiate between the classes. Also, the logistic Regression is a linear model, and if the relationship between the features and the target variable is non-linear, it may struggle to capture the patterns, leading to lower accuracy.

Poor performance of 'saga' solver

- The saga solver might be slower and less efficient on this dataset because it's better suited for very large datasets or datasets with sparse features. The penguins dataset is relatively small, and saga may not be necessary.

1.8 Classification Accuracy with 'liblinear' Solver



```python
X_train , X_test , y_train , y_test = train_test_split(X , y ,test_size =0.2 , random_state =42)

#Train the logistic regression model . Here we are using liblinear to learn weights .
logreg = LogisticRegression(solver ='liblinear')
logreg.fit( X_train , y_train )

# Predict on the testing data
y_pred = logreg.predict( X_test )

# Evaluate the model
accuracy = accuracy_score( y_test , y_pred )
print ("Accuracy :", accuracy )
print ( logreg.coef_ , logreg.intercept_ )
```

- Accuracy = 1.0

1.9 Why 'liblinear' Performs Better Than 'saga'

- 'liblinear' is specifically designed for small to medium-sized datasets and is optimized for binary classification problems like this problem. saga, on the other hand, is designed to handle large datasets. Since penguins dataset is relatively small, the saga solver might be inefficient, and liblinear outperforms it.

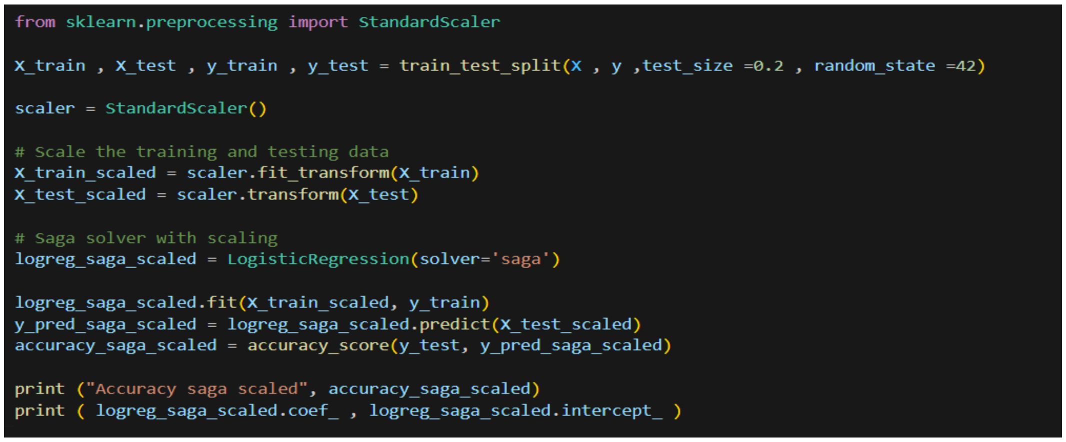
- Also, 'liblinear' uses the coordinate descent algorithm, which converges more quickly and stably for small datasets. saga is a stochastic optimization algorithm. In smaller datasets, stochastic methods can exhibit higher variance during optimization. So, liblinear's simplicity, lower variance in optimization, and quicker convergence make it perform better than saga on this dataset.

- Binary classification tasks and one-vs-rest (OvR) multiclass problems are efficiently handled by the liblinear solver, which is particularly suited for smaller datasets. In contrast, for multinomial classification, especially on larger datasets, the saga solver is more appropriate. However, saga may not perform as well on smaller datasets, where liblinear is generally preferred for simpler classification tasks.

1.10 Comparing 'liblinear' and 'saga' Solvers with Feature Scaling

Without Feature Scaling:

- Accuracy with ’liblinear’ : 1.0
- Accuracy with ’saga’ : 0.5813953488372093



```python
from sklearn.preprocessing import StandardScaler

X_train , X_test , y_train , y_test = train_test_split(X , y ,test_size =0.2 , random_state =42)

scaler = StandardScaler()

# Scale the training and testing data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Saga solver with scaling
logreg_saga_scaled = LogisticRegression(solver='saga')

logreg_saga_scaled.fit(X_train_scaled, y_train)
y_pred_saga_scaled = logreg_saga_scaled.predict(X_test_scaled)
accuracy_saga_scaled = accuracy_score(y_test, y_pred_saga_scaled)

print ("Accuracy saga scaled", accuracy_saga_scaled)
print ( logreg_saga_scaled.coef_ , logreg_saga_scaled.intercept_ )
```

After applying Standard Scaler:

Both the liblinear and saga solvers achieve similar accuracy:

- liblinear Accuracy (scaled) : 0.9767
- saga Accuracy (scaled) : 0.9767

So, when feature scaling is applied, both solvers perform almost identically, achieving high accuracy and similar model parameters. As Feature scaling standardizes the features to have a mean of 0 and a standard deviation of 1, ensuring that all features contribute equally to the optimization process, this allows the solver to learn better weights and improves model convergence.

But we can see a significant accuracy improvement in saga solver because the saga solver is a stochastic gradient descent-based algorithm, which is more sensitive to feature scaling because it involves updating weights iteratively. Without scaling, features with larger magnitudes dominate the gradient updates, making convergence slower and less accurate. The liblinear solver is based on coordinate descent and is less sensitive to the magnitudes of features, but even it can benefit from feature scaling for more balanced updates across all features.

1.11 Problem with Code in Listing 3 and How to Fix It

Problems encountered in the code:

- The features “island” and “sex” in the dataset are categorical, meaning they represent distinct categories rather than numerical values. To use these variables in the Logistic Regression model, they need to be converted into a numerical format. This transformation ensures that the model can effectively interpret and learn from these features.

Solution for the problem encountered:

- Use One-Hot Encoding or Label encoding to convert categorical variables into a format that can be provided to machine learning algorithms to do a better job in prediction.

1.12 Scaling After Label Encoding for Categorical Features: Correct or Not?

About the approach mentioned in the question:

- No, applying Standard Scaling or Min-Max Scaling directly after Label Encoding is not a correct approach when dealing with categorical features. Label Encoding assigns arbitrary numerical values (like 0, 1, 2) to the categories, but these numbers do not represent any inherent order or relationship between the categories. Applying scaling methods like Standard Scaling or Min-Max Scaling assumes that the values are continuous and have some mathematical relationship, but in this case, the numbers 0, 1, and 2 are just categorical labels with no numerical meaning.

Correct Approach:

- Instead of scaling a label-encoded categorical feature, we can use One-Hot Encoding for categorical data. This method properly handle categorical variables without distorting the relationships between them. This ensures that this machine learning model treats categorical data appropriately.

Question 2

This data set includes following variables

- x_1 = Number of hours studied.
- x_2 = Undergraduate GPA.
- y = Whether the student received an A+ in the class.

The estimated logistic regression coefficients are given as:

$w_0 = -5.9$, $w_1 = 0.06$, $w_2 = 1.5$

1.

$P(y_i = 1|x_i,w) = \text{sigm}(w_0 + w_1x_1 + w_2x_2) = \text{sigm}(-5.9 + 0.06*50 + 1.5*3.6)$

$$= \frac{1}{1 + e^{-(-5.9+0.06(50)+1.5(3.6))}} =0.924$$

2.

$P(y_i = 1|x_i,w) = 0.6 = \text{sigm}(w_0 + w_1x_1 + w_2x_2) = \text{sigm}(-5.9 + 0.06*H + 1.5*3.6)$

$=1/(1+ e^{-(-0.5+0.06H)})$

Therefore, $H = (0.5 – \ln(2/3))/ 0.06 = 15.091 \approx 15.1$

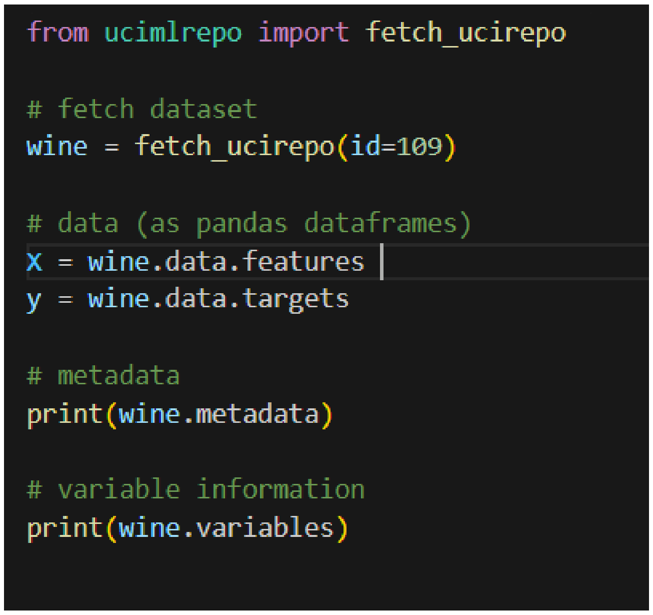
Here, H is the number of hours that was studied by the student.

So, the student needs to study approximately 15.08 hours to achieve a 60% chance of receiving an A+.

2. Logistic regression on real world data

2.1 Choosing a Dataset for Logistic Regression

I have chosen ‘Wine’ data set

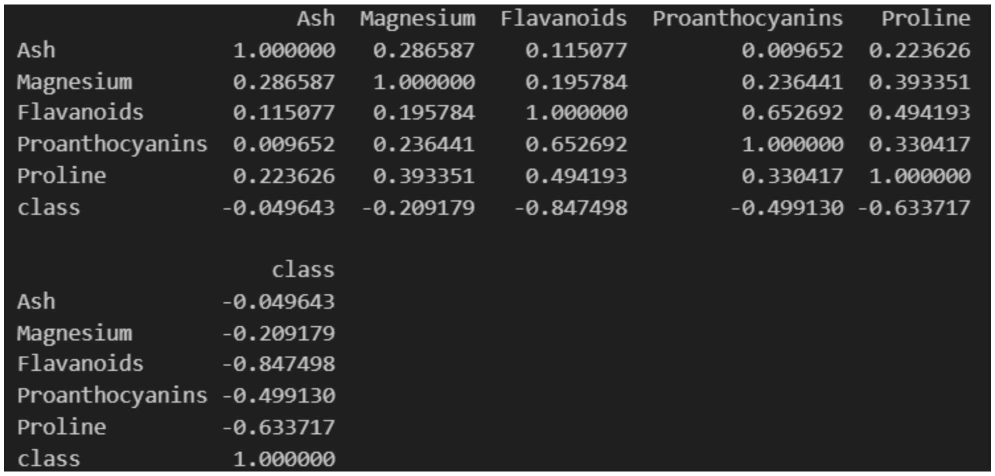


```
from ucimlrepo import fetch_ucirepo

# fetch dataset
wine = fetch_ucirepo(id=109)

# data (as pandas dataframes)
X = wine.data.features
y = wine.data.targets

# metadata
print(wine.metadata)

# variable information
print(wine.variables)
```

2.2 Correlation Matrix and Pair Plot Analysis

I have chosen 'Ash', 'Magnesium', 'Flavanoids', 'Proanthocyanins', 'Proline' as the 5 features for analysis.

Here ‘**class'** is the target variable.

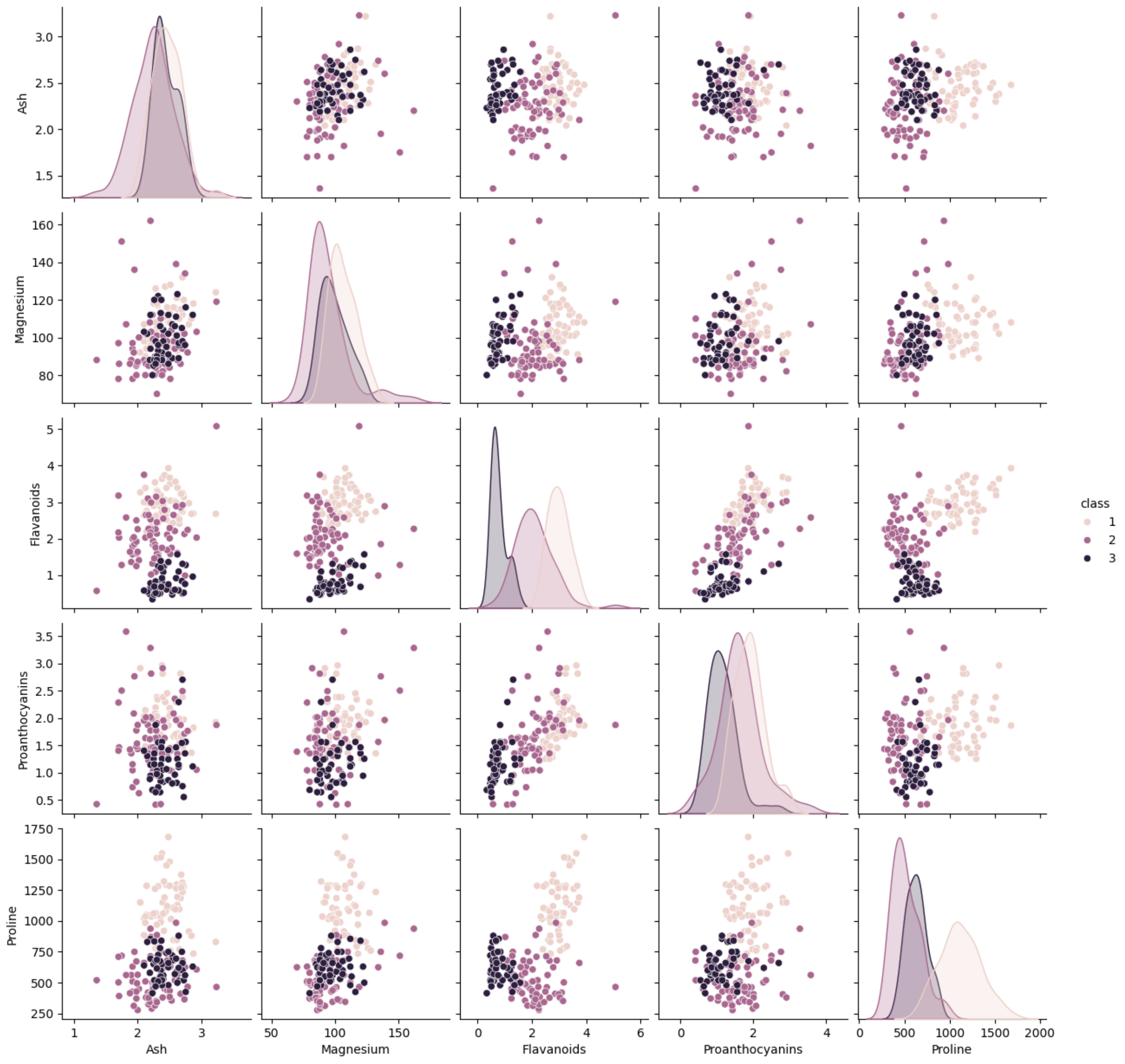


```
                      Ash  Magnesium  Flavanoids  Proanthocyanins   Proline
Ash              1.000000   0.286587    0.115077         0.009652  0.223626
Magnesium        0.286587   1.000000    0.195784         0.236441  0.393351
Flavanoids       0.115077   0.195784    1.000000         0.652692  0.494193
Proanthocyanins  0.009652   0.236441    0.652692         1.000000  0.330417
Proline          0.223626   0.393351    0.494193         0.330417  1.000000
class           -0.049643  -0.209179   -0.847498        -0.499130 -0.633717

                    class
Ash             -0.049643
Magnesium       -0.209179
Flavanoids      -0.847498
Proanthocyanins -0.499130
Proline         -0.633717
class            1.000000
```

Correlation Matrix Insights:

- Ash and Magnesium show a moderate positive correlation (0.29), suggesting that as the Ash content increases, Magnesium levels tend to increase as well, but not strongly.
- Flavanoids and Proanthocyanins have a strong positive correlation (0.65), indicating a significant relationship between these two variables. This could be because both are related to phenolic compounds in wine.
- Magnesium and Proline also exhibit a moderate positive correlation (0.39), meaning wines with higher magnesium content also tend to have higher Proline levels.
- Flavanoids and Proline have a moderate correlation (0.49), which is an interesting observation as both compounds influence the flavor profile of wine.
- Class shows negative correlations with most of the selected features, especially:
    - Flavanoids (-0.85): Wines with higher flavanoid content tend to be of a specific class.
    - Proline (-0.63): Wines with higher Proline content are more likely to belong to certain classes, suggesting that these are distinguishing features between wine types.

Pair Plot Observations:



- Flavanoids and Proanthocyanins exhibit a strong positive relationship with minimal overlap between classes, particularly for Class 1, which appears to have the highest concentration of these compounds.

- Proline shows a clear distinction between classes, with Class 3 wines having higher Proline concentrations.

- Magnesium has less class separation compared to other variables but still shows some clustering, particularly between Class 1 and the others.

- Ash appears to have the weakest correlation with class labels and does not seem to separate the classes clearly in the pair plot

Overall, by the selected 5 features for the analysis, Flavanoids, Proline, and Proanthocyanins seem to be the most significant features in differentiating wine classes, while Ash and Magnesium provide less distinction across classes. These observations can help in feature selection for classification or further analysis in wine datasets.

2.3 Fitting and Evaluating the Logistic Regression Model

I have got the results for **Accuracy, Precision, Recall, F1-Score** and **confusion Matrix**.

- Accuracy (80.56%): the model correctly predicted the wine class 80.56% of the time, which is a good result for a basic logistic regression model.

- Precision (80.58%): this means that when the model predicted a specific class (Class 1, 2, or 3), about 80.58% of the predictions were correct.

- Recall (80.56%): A recall of 80.56% means the model was able to correctly identify around 80.56% of the actual instances of each wine class.

- F1-Score (80.47%): An F1-score of 80.47% means your model performs consistently across both precision and recall.

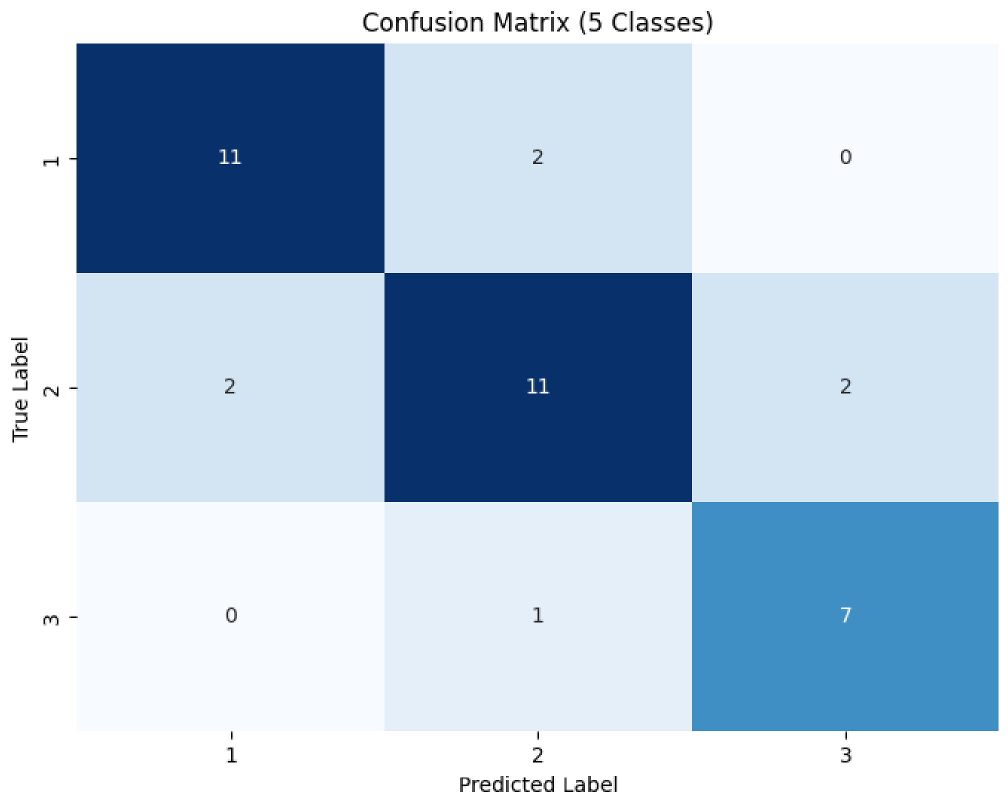
- Confusion Matrix

```
[[11  2  0]

[ 2 11  2]

[ 0  1  7]]
```

From the confusion matrix,

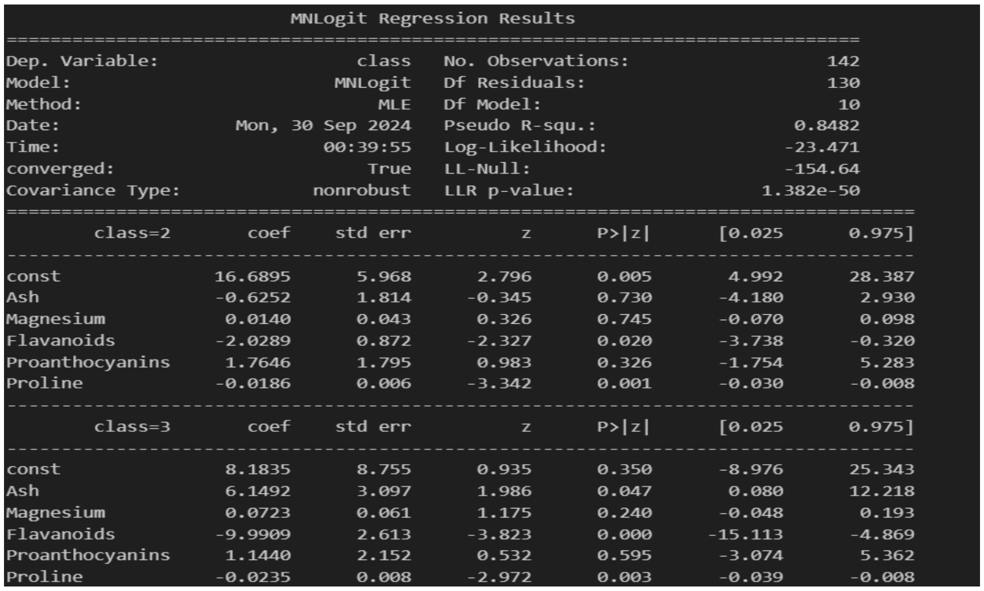
- Class 1 (First row): 11 out of 13 samples were correctly classified, with 2 being misclassified as Class 2.
- Class 2 (Second row): 11 out of 15 samples were correctly classified, with 2 misclassified as Class 3 and 2 as Class 1.
- Class 3 (Third row): 7 out of 8 samples were correctly classified, with 1 misclassified as Class 2.

This implemented logistic regression model is performing quite well with the selected five features. An accuracy of 80.56% suggests that the model can reliably predict the correct class in most cases, and the precision, recall, and F1-score are also consistent, indicating balanced performance.



2.4 P-Values and Feature Selection with Statsmodels

We selected a 95% confidence interval and designated Class 1 as the reference class.



```
                          MNLogit Regression Results
==============================================================================
Dep. Variable:                  class   No. Observations:                  142
Model:                        MNLogit   Df Residuals:                      130
Method:                           MLE   Df Model:                           10
Date:                Mon, 30 Sep 2024   Pseudo R-squ.:                  0.8482
Time:                        00:39:55   Log-Likelihood:                -23.471
converged:                       True   LL-Null:                       -154.64
Covariance Type:            nonrobust   LLR p-value:                 1.382e-50
===================================================================================
        class=2       coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------------
const              16.6895      5.968      2.796      0.005       4.992      28.387
Ash                -0.6252      1.814     -0.345      0.730      -4.180       2.930
Magnesium           0.0140      0.043      0.326      0.745      -0.070       0.098
Flavanoids         -2.0289      0.872     -2.327      0.020      -3.738      -0.320
Proanthocyanins     1.7646      1.795      0.983      0.326      -1.754       5.283
Proline            -0.0186      0.006     -3.342      0.001      -0.030      -0.008
-----------------------------------------------------------------------------------
        class=3       coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------------
const               8.1835      8.755      0.935      0.350      -8.976      25.343
Ash                 6.1492      3.097      1.986      0.047       0.080      12.218
Magnesium           0.0723      0.061      1.175      0.240      -0.048       0.193
Flavanoids         -9.9909      2.613     -3.823      0.000     -15.113      -4.869
Proanthocyanins     1.1440      2.152      0.532      0.595      -3.074       5.362
Proline            -0.0235      0.008     -2.972      0.003      -0.039      -0.008
```

Features that can be discarded:

- Ash: Not significant in class 2 (Class 2: p = 0.730).
- Magnesium: Not significant in both classes (Class 2: p = 0.745; Class 3: p = 0.240).
- Proanthocyanins: Not significant in both classes (Class 2: p = 0.326; Class 3: p = 0.595).

Significant features to keep:

- Ash: Significant in class 3 (Class 3 : p= 0.047)
- Flavanoids: Significant in both classes, indicating a strong relationship with the outcome (Class 2: p = 0.020; Class 3: p = 0.000).
- Proline: Also significant in both classes, suggesting its importance in the analysis (Class 2: p = 0.001; Class 3: p = 0.003).

Overall, the analysis suggests that while features like Ash, Magnesium, and Proanthocyanins can be discarded due to their lack of significance in certain classes, Flavanoids and Proline show strong associations with the outcome in both classes and should be retained for further analysis.

3. Logistic regression First/Second-Order Methods

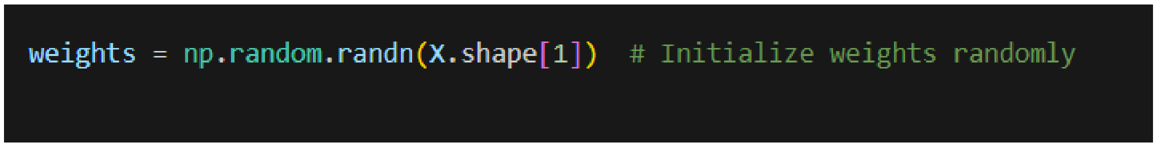
3.1 Data Generation for Logistic Regression

Generate data by using listing 4.

3.2 Batch Gradient Descent: Weight Update and Initialization

Method Used for Weight Initialization:

The weights are initialized randomly using the following line:



```
weights = np.random.randn(X.shape[1])  # Initialize weights randomly
```

This initialization creates an array of weights with random values drawn from a **standard normal distribution** (mean of 0 and standard deviation of 1).

Reason for Selection:

- Random initialization ensures that the weights are not all identical, preventing **symmetry** during training. If all weights were initialized to the same value (like zeros), they would remain the same during the update steps. This is because each weight would receive the same gradient and hence get updated identically. Random values break this symmetry, allowing each weight to learn independently.
- Random initialization provides a good starting point for the optimization algorithm (gradient descent). This method generally helps the model converge faster compared to initializing weights with zeros or very large values.

3.3 Selection of Loss Function

The loss function used is the **Cross-Entropy error function.**

$$J(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1 - y_n)\log(1 - \hat{y}_n)\right]$$

where $\hat{y}_n \equiv g(\mathbf{w}\cdot\mathbf{x}_n) = 1/(1 + e^{-\mathbf{w}\cdot\mathbf{x}_n})$, with $g(z)$ the logistic function as before.

y_n represents the true labels.

N is the number of samples.

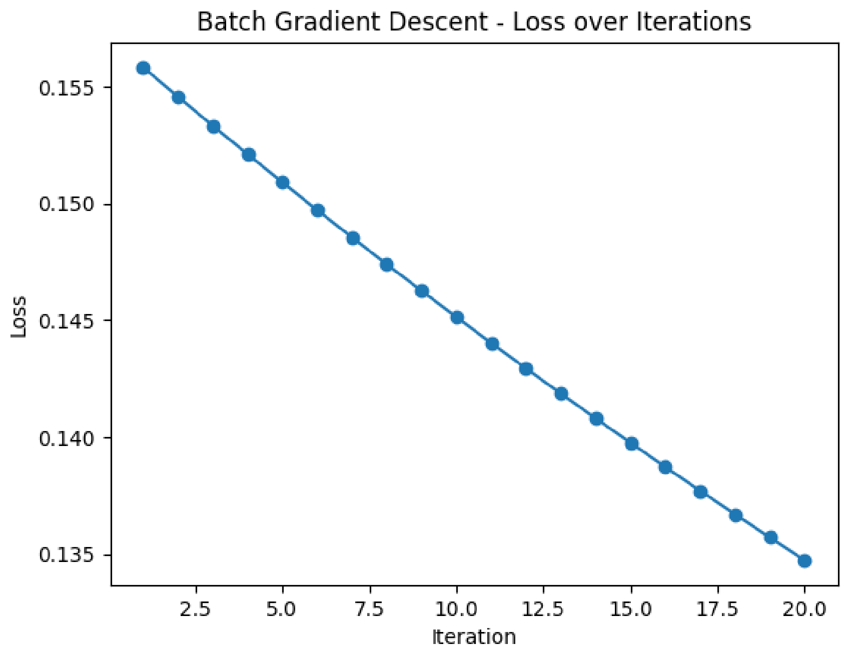
```
return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
```

Reason for Selection:

- Cross-entropy loss is ideal for binary classification tasks as it measures the difference between the actual class labels and the predicted probabilities. It effectively penalizes incorrect predictions, especially when the model is confident but wrong.

- Also, this loss function provides a smooth gradient, which helps in optimizing the weights efficiently during gradient descent, leading to faster convergence.

3.4 Plotting Loss vs. Iterations (Batch Gradient Descent)



3.5 Stochastic Gradient Descent: Weight Update

Method Used for Weight Initialization:

In the Stochastic Gradient Descent (SGD) implementation, the weights are initialized randomly using the following line:

```
weights = np.random.randn(X.shape[1])  # Random initialization of weights
```

This initializes the weights by drawing random values from a **standard normal distribution** (mean of 0 and standard deviation of 1).

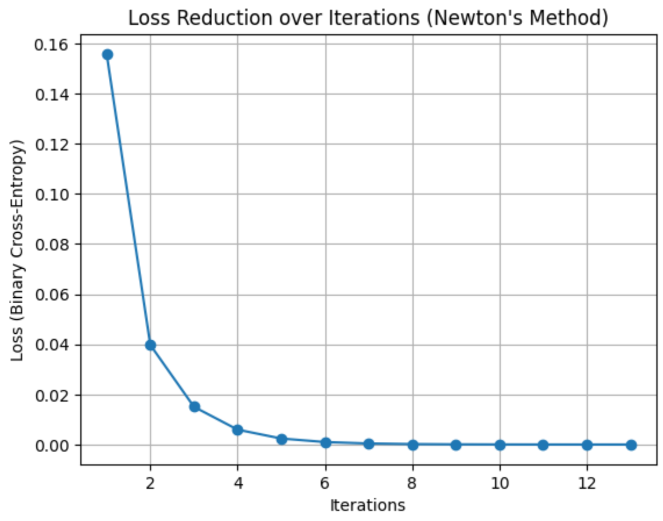
Reason for Selection:

- Initializing the weights randomly ensures that all neurons or features start with different initial values, breaking symmetry. If the weights were initialized with the same value (e.g., all zeros), the model would treat all features similarly, and the gradients would remain the same for each weight, making learning inefficient.
- Random initialization helps avoid scenarios where the gradient descent algorithm gets stuck in flat regions or saddle points early on, which can occur when using zero initialization.

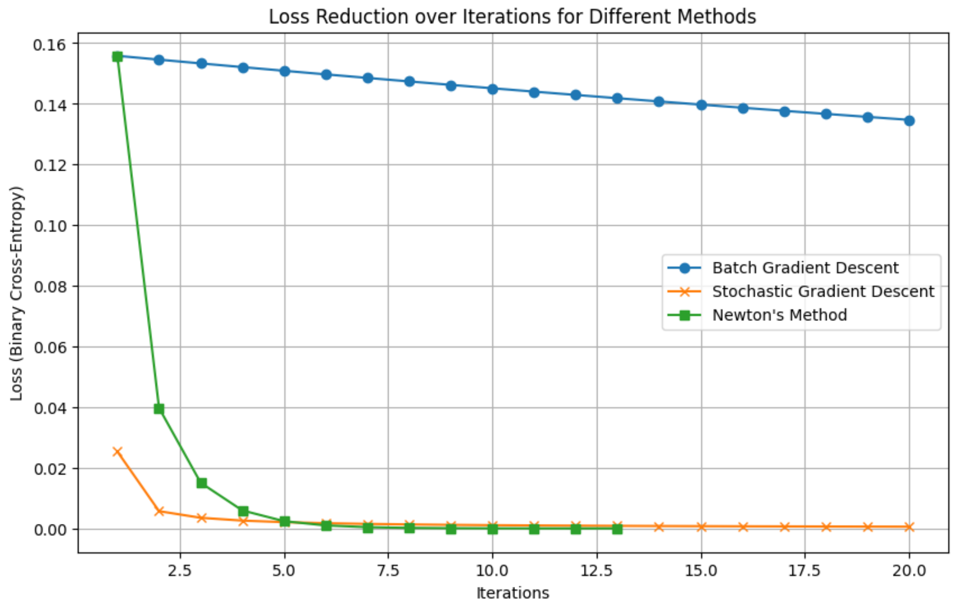
3.6 Newton's Method: Weight Update

Implementation of the Newton’s Method.

3.7 Plotting Loss vs. Iterations (Newton's Method)



3.8 Comparison of Loss: Gradient Descent, Stochastic Gradient Descent, and Newton’s Method



Comment on the Results:

- The graph clearly shows the loss reduction for each method over the specified iterations. All three methods—Batch Gradient Descent, Stochastic Gradient Descent (SGD), and Newton's Method—successfully reduce the binary cross-entropy loss over time.

- Batch Gradient Descent:

  Initial Behavior: The loss starts relatively high and decreases rapidly in the first few iterations, indicating that this method is effective in making significant weight adjustments.

  Convergence: The loss stabilizes at a lower value but does not drop as low as the other two methods, suggesting that while it is stable, it may require more iterations or a better learning rate to reach a minimal loss.

- Stochastic Gradient Descent:

  Loss Fluctuations: The SGD path is noticeably more erratic, with losses showing significant variation between iterations. This behavior is expected as SGD updates weights based on single data points, introducing noise.

  Convergence: The loss does decrease but remains higher than that of Newton's Method, indicating that while SGD is quick to make adjustments, it may struggle to consistently reduce loss without more iterations. There's a general downward trend, indicating that the algorithm is converging.

- Newton's Method:

  Fast Convergence: Newton's Method displays a rapid decrease in loss, particularly in the initial iterations. The loss quickly approaches its minimum, showing the method's effectiveness at leveraging second-order derivative information.

  Stability: The loss values quickly level out, indicating that the method finds a local minimum efficiently. This is evidenced by the relatively low loss achieved by the end of the iterations compared to the other methods.

Conclusion:

- Newton's Method outperforms both Gradient Descent methods (Batch Gradient Descent and Stochastic Gradient Descent) in terms of loss reduction and speed of convergence, achieving the lowest loss with the fewest iterations. However, it is computationally heavier and may not be practical for very large datasets.

- Also, Batch Gradient Descent provides stability and is suitable for smaller datasets, while SGD is more adaptable for larger datasets but may require tuning of parameters to achieve optimal performance.

3.9 Approaches to Decide Number of Iterations

Gradient Descent:

1.Convergence Criteria

- Loss Change Threshold: Define a threshold for the change in loss between iterations. The training process can be terminated if the absolute difference in loss falls below a specified epsilon value:

  $$|J(W_{t+1}) - J(W_t)| < \epsilon$$

  This criterion ensures that the algorithm stops when further iterations yield minimal improvement, indicating convergence.

- Gradient Magnitude: Set a condition based on the magnitude of the gradient. If the gradient norm drops below a designated delta, it suggests that updates to the weights are no longer significant:

  $$\|\nabla J(W)\| < \delta$$

This can effectively indicate when the optimization process has reached a local minimum.

2.Early Stopping Using a Validation Set

- Validation Monitoring: Start with a maximum predetermined number of iterations and evaluate the model on a separate validation dataset after each iteration. If the validation loss fails to improve for a set number of consecutive iterations, the training can be halted. This method not only helps in avoiding overfitting but also allows the determination of the optimal number of iterations required for effective convergence.

Newton’s Method:

1.Hessian Condition

- Condition Number Monitoring: Utilize the condition number of the Hessian matrix as an adaptive stopping criterion. A high condition number may indicate that the optimization is not progressing efficiently, leading to potential divergence. In such cases, the process should be stopped to avoid unnecessary computations.

- Weight Change Monitoring: Track the change in weights between iterations. The training can be terminated if the difference becomes sufficiently small:

  $$\|W_{t+1} - W_t\| < \epsilon$$

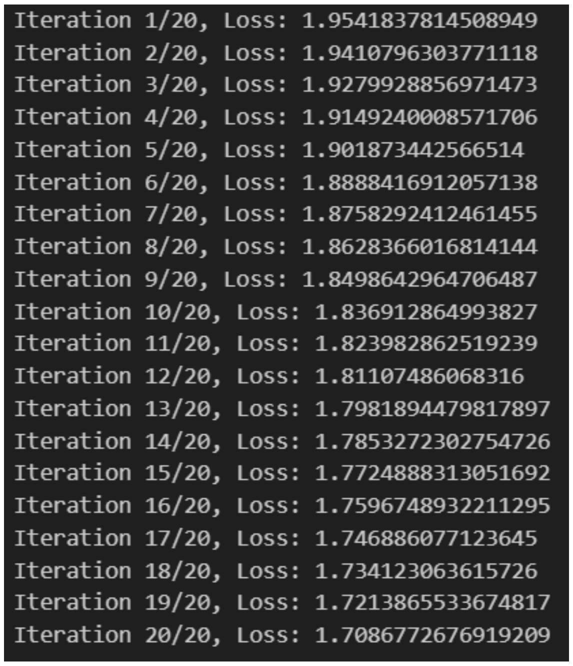
  This approach ensures that the algorithm ceases when weight updates are no longer substantial.

2.Maximum Iterations with Improvement Assessment

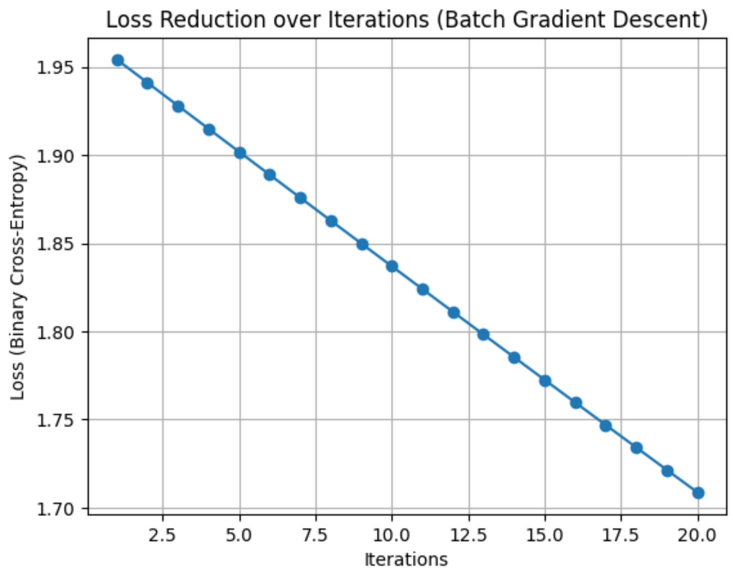
- Setting a Cap on Iterations: Establish a maximum number of iterations for the optimization process. However, after a specific number of iterations, evaluate the loss improvement. If the enhancement falls below a certain percentage (for instance, less than 1%), the iterations can be halted. This strategy helps in conserving computational resources when further updates do not significantly impact the outcome.

3.10 Convergence Analysis for New Centers in Gradient Descent

Convergence Behavior Analysis after changing the centers.



```
Iteration 1/20, Loss: 1.9541837814508949
Iteration 2/20, Loss: 1.9410796303771118
Iteration 3/20, Loss: 1.9279928856971473
Iteration 4/20, Loss: 1.9149240008571706
Iteration 5/20, Loss: 1.901873442566514
Iteration 6/20, Loss: 1.8888416912057138
Iteration 7/20, Loss: 1.8758292412461455
Iteration 8/20, Loss: 1.8628366016814144
Iteration 9/20, Loss: 1.8498642964706487
Iteration 10/20, Loss: 1.836912864993827
Iteration 11/20, Loss: 1.823982862519239
Iteration 12/20, Loss: 1.81107486068316
Iteration 13/20, Loss: 1.7981894479817897
Iteration 14/20, Loss: 1.7853272302754726
Iteration 15/20, Loss: 1.7724888313051692
Iteration 16/20, Loss: 1.7596748932211295
Iteration 17/20, Loss: 1.746886077123645
Iteration 18/20, Loss: 1.734123063615726
Iteration 19/20, Loss: 1.7213865533674817
Iteration 20/20, Loss: 1.7086772676919209
```



Analyzing the convergence behavior:

Initial Loss Values:

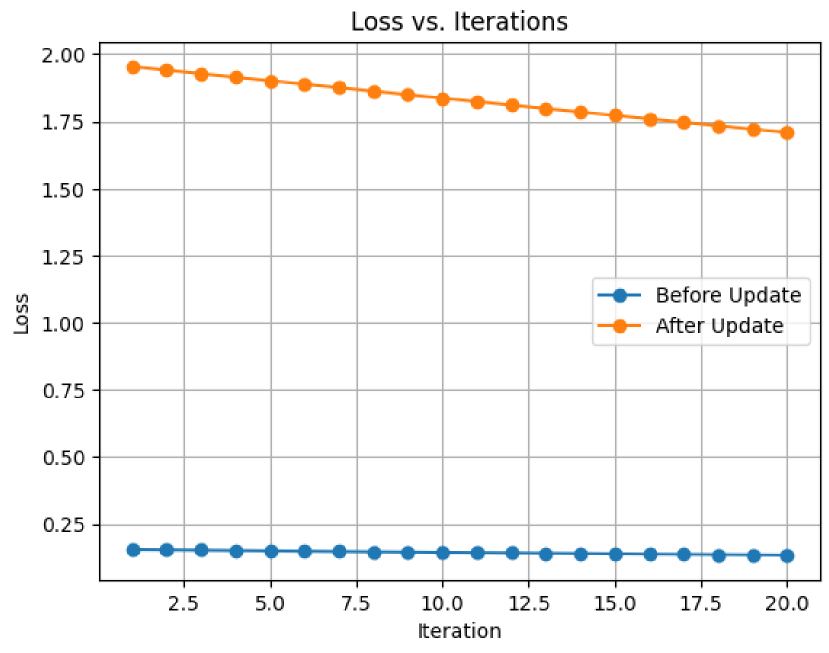
- The updated centers have resulted in an initial loss of approximately 1.95. This suggests that the model's starting point (initialized weights) is relatively far from the optimal weights required to fit the new data configuration. The higher starting loss indicates that the initial predictions are not closely aligning with the actual labels.

Loss Reduction:

- The printed loss values show a consistent decrease over the iterations, moving from 1.954 to 1.708 by the end of the 20 iterations. This consistent reduction indicates that the batch gradient descent algorithm is effectively learning and adjusting the weights to minimize the error between the predicted and actual labels.

Rate of Convergence:

- The rate of loss reduction can vary. In this case, the loss decreases steadily, suggesting that the learning rate and the configuration of the data are suitable for the gradient descent algorithm. If the centers had been positioned in a way that significantly misaligned with the initial weights, convergence could have been slower, resulting in smaller reductions in loss per iteration.



- The "Before Update" curve shows a slower convergence, as evidenced by the gradual loss decrease, while the "After Update" scenario demonstrates a sharper decline in loss, suggesting that the updated centers were more aligned with the true data distribution.

Final Loss Value:

- The final loss of approximately 1.708 indicates that while the algorithm has improved the fit to the data, there may still be room for further optimization. This suggests that more iterations or a different learning rate might yield better results.

Conclusion about the convergence behavior.

The convergence behavior of the Batch Gradient Descent algorithm is notably influenced by the configuration of the dataset, particularly the positions of the centers. The updated centers appear to have created a higher initial loss, but the algorithm has shown a reliable ability to reduce the loss over iterations.

4. References:

[1] “Cross-entropy,” *Wikipedia*, Aug. 14, 2023. https://en.wikipedia.org/wiki/Cross-entropy

[2] GeeksforGeeks, “Newton’s method in Machine Learning,” *GeeksforGeeks*, Apr. 23, 2024. https://www.geeksforgeeks.org/newtons-method-in-machine-learning/ (accessed Sep. 30, 2024).

[3]ArnavR, “Scikit-learn solvers explained,” *Medium*, Mar. 22, 2022. https://medium.com/@arnavr/scikit-learn-solvers-explained-780a17bc322d